CE323 - Advanced Embedded Systems Design

Assignment Report
Home Alarm System

Akshay Gopinath
Registration Number: 2005614

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

This report documents a design of a home alarm system on an Embedded platform. The target embedded device is the ARM mbed LPC1768 development board which houses a 32-bit Cortex-M3 Microcontroller. The software design used is scheduler based, where tasks are configured to run at set refresh rates, allowing for clean, extendable, modular and flexible code.

1.1. Requirments Form

Name: Home Alarm System

Purpose: To prevent uninvited house intrusion by detecting sensor activation within the various zones.

Inputs: Keypad to enter password, Normally Open/Closed sensors/switches at each zone.

Outputs: LED to display the system status, LCD screen for user interface.

Function Specifications: The system has 6 states, unset, exit, set, entry, alarm and report.

- **Unset State:** Activation of sensors should not cause the alarm LED to blink, and entry of the correct passcode will cause a transition to the set state. Entry of wrong passcode will transition to alarm state.
- **Exit State:** There is a configurable time interval (exit period) for example 1 minute for evacuation. The alarm LED will be blinking. Entry of correct passcode will transition back to unset state. Transition to alarm state occurs when incorrect code is entered three times or if any sensor is activated. If all sensors are inactive after the exit period expires, the system transitions to the entry state.
- **Set State:** Activation of entry/exit zone sensor causes a transition to the entry state. Activation of other sensors will make the system transition to the alarm state.
- **Entry State:** The entry period is a configurable time limit, for example 1 minute. The alarm LED will blink. The correct passcode will change the system back to unset state. Activation of any sensor or if the correct password is not entered within the time limit, the system transitions to the alarm state.
- **Alarm State:** Alarm LED should be on all the time and switch off after 2 minutes. If incorrect password is entered, the system stays in alarm state, else it transitions to report state.
- **Report State:** The LCD screen displays which zones have been triggered (the error code). And the system can be cleared (transition back to unset state) when the enter button is pressed.

Performance: The tasks refresh rates in the system have been tuned for optimal performance. The is LCD updated every 200ms, the keypad polling refresh rate is set to 100ms. The sensor/switch task is ran at a rate of 100ms.

Manufacture costs: Less than £100 including the Microcontroller, LCD screen, keypad and sensors/switches.

Physical size/weight: The sensors/switches should be sized well to comfortably fit into doors and windows. The user interface (keyboard and LCD) should fit on a house dashboard.

Project Constraints: Due to the hardware available, switches will represent sensor activation and an LED will represent the alarm. Figure 4 in the appendix shows the hardware used for the project.

Design Parameters: The software has various configurable paramaters such as exit state and entry state time limit, LCD refresh rate and Keypad poll rate, alarm state LED on time, password, sensor poll rate.

2. STATE MACHINE DIAGRAM

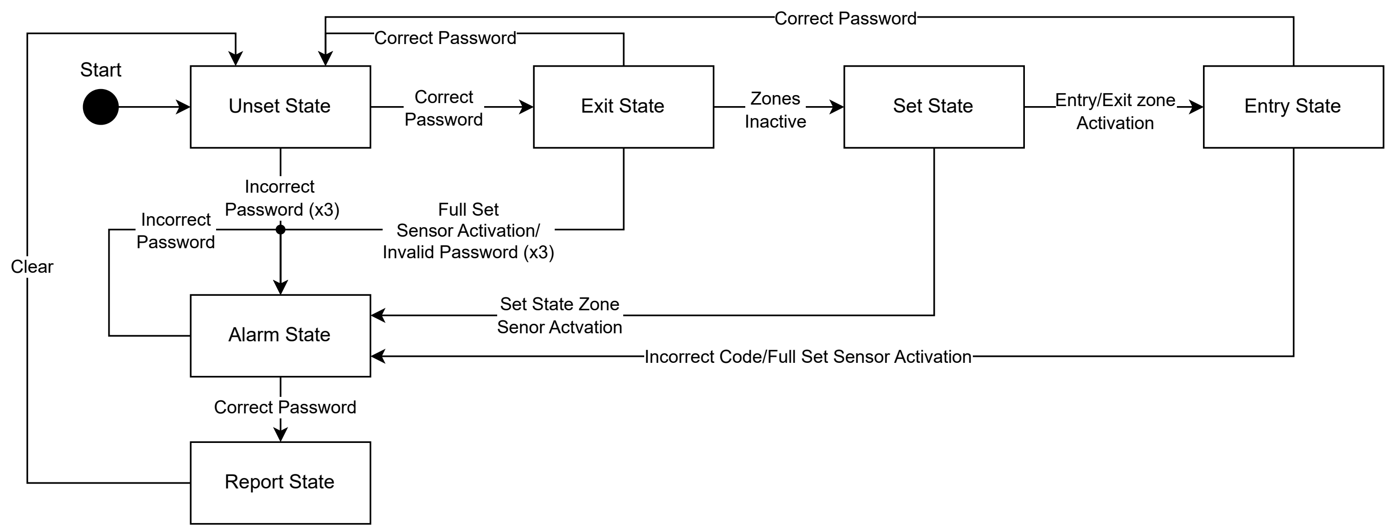

FIG. 1: *State Machine Diagram*

Figure 1 above represents the state machine diagram of the home alarm system. The system has 6 states, which are: Unset State, Exit State, Set State, Entry State, Alarm State, Report State. The starting state is the Unset state. Since the system always loops back to the Unset State, there is no ending state.

In the Unset State, the system will not react to any sensor activation. If the wrong password is entered three times, the system jumps to the alarm state. The entry of the correct four digit password will result in a transition to the Exit State.

During the Exit State, the user has an ‘exit period’ during which they can evacuate their home. The entry period in the software is set to one minute, and is configurable in the code. The system changes to a different state depending on certain conditions. If any of the eight sensors is activated, the alarm system will jump to the alarm state, and also if the incorrect password is entered three times. Within the exit period, if the correct password is entered, the system transitions back to the unset state. If the exit period expires and all sensors are inactive, the system enters the set state. The alarm LED will be blinking in this state.

Within the set state, if the entry/exit zone sensor is activated, the system enters the Entry State, else if any of the other 7 sensors is activated, the system will move to the Alarm State.

In the Entry State, there is a period of time when the user can gain access to their home, so that the user can unset the alarm. The ‘entry period’ (the time limit) is set to one minute, which is also configurable in the code. If the correct 4 digit passcode is entered, the system transitions back to the unset state. The system enters the alarm state under two conditions. The first when the user fails to enter the correct password within in the time limit. The second when the any of the sensors are activated. In this state, the alarm LED is blinking.

In the alarm state, the alarm LED is on all the time and after 2 minutes, it switches off. If the correct password is entered, the system enters the report state, else it stays in the alarm state.

In the report state, the LCD shows the error code. The error code can either be the zone numbers where the sensors were triggered, or the incorrect password. And the user can clear the system by pressing a button on the keypad. Once the system is cleared, the system moves back to the starting state, which is the Unset State.

3. CLASS DIAGRAM

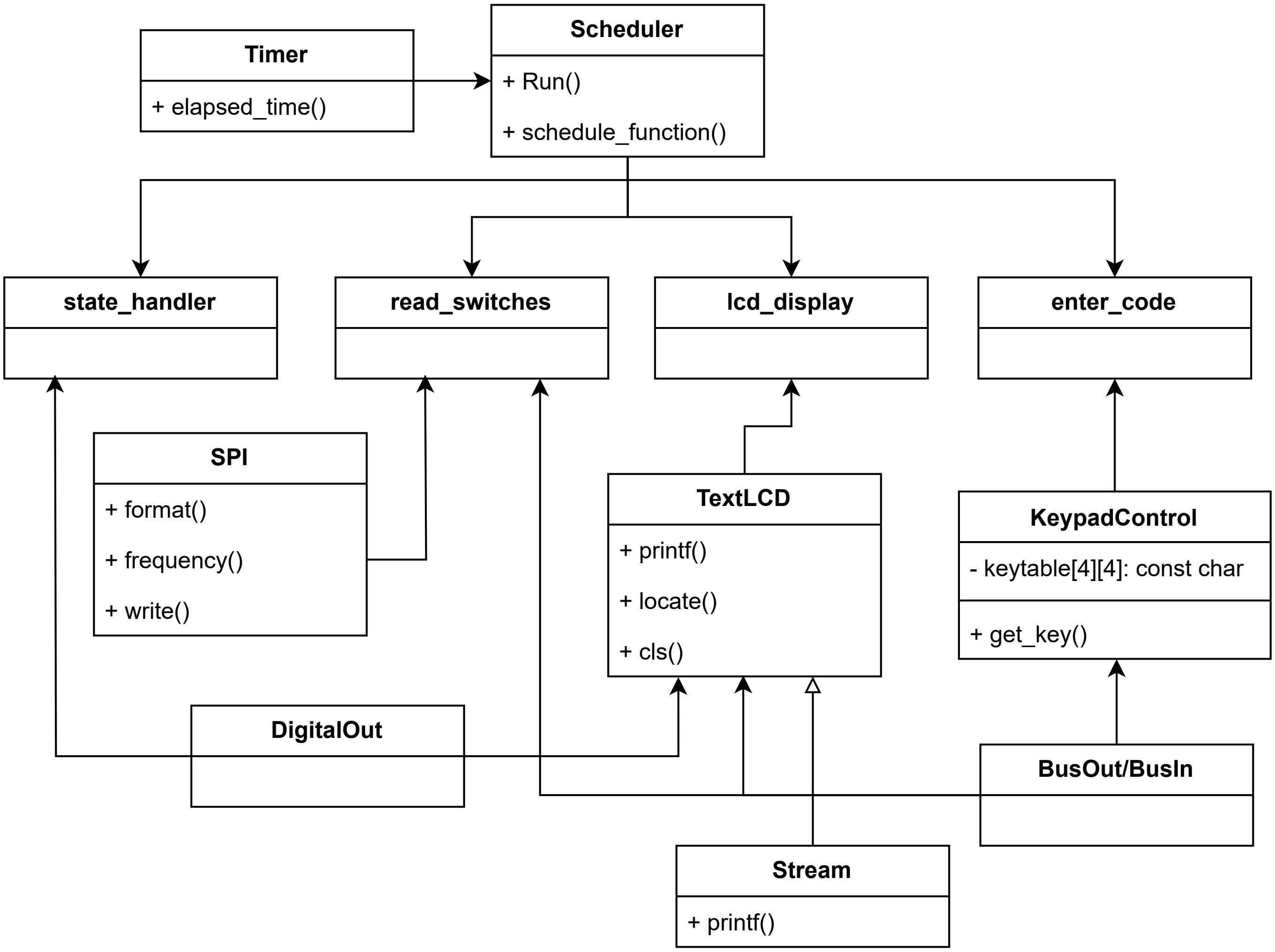

FIG. 2: *Class Diagram*

Figure 2 above shows the class dependency diagram for the home alarm system. The main heart of the software is the `Scheduler` class, which controls and runs all the operations of the system (for example displaying to LCD). The scheduler class depends on the internal timer of the mbed, which is implemented by the `Timer` class. The `Timer` class is provided by the mbed framework library. The Scheduler controls four free form functions: `state_handler`, `read_switches`, `lcd_display` and `enter_code`. Since these methods act independently because of the scheduler, they are treated like objects for this diagram. The `state_handler` task depends on the `DigitalOut` class from the mbed framework, as the `state_handler` also controls the alarm LED. The `read_switches` task depends on `SPI`, `BusOut` and `BusIn`. The `BusOut` class is used as the chip select to select the switches and `BusIn` is used to receive the switch readings. Serial Peripheral Interface (SPI) protocol is used to communicate with the red/green LEDs on the hardware which is set according to the current switch reading.

The `TextLCD` library is a third party library to drive the LCD. `TextLCD` inherits of a class called `Stream` which is an internal class in the mbed framework to handle standard input and output (`stdin` and `stdout`). This library also depends on `BusOut`, `BusIn` and `DigitalOut`. The `lcd_display` task depends on the `TextLCD` library.

The `KeypadControl` class was created to drive the keypad on the hardware for user input. And this class also depends on the `BusOut` and `BusIn` from the mbed library. The `enter_code` task uses the `KeypadControl` class object to take user input.

4. SEQUENCE DIAGRAM

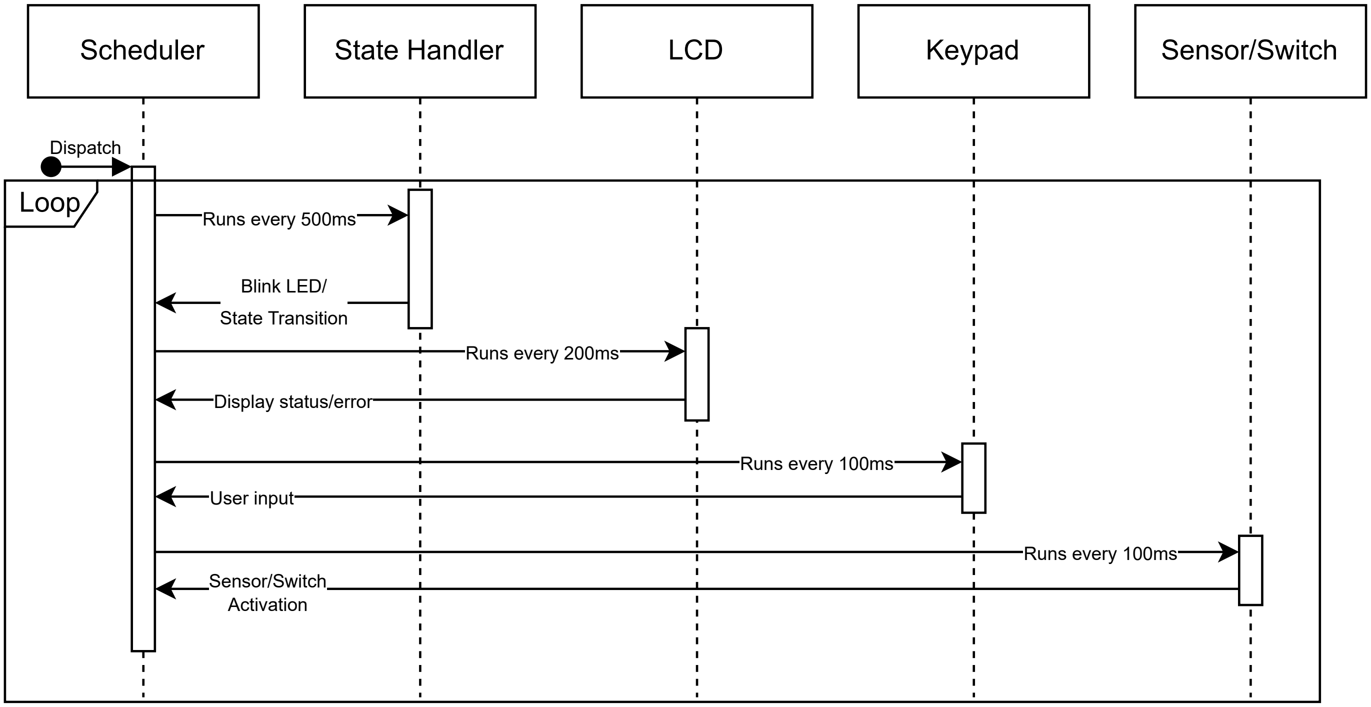

FIG. 3: *Sequence Diagram*

Figure 3 above shows the sequence diagram of the home alarm system. The sequence diagram depicts the order and timeline of the system behaviour. Since the software design is based on a task based scheduling system, the sequence diagram is represented by the individual tasks that are spun in the main scheduling loop. The running tasks are placed in the order at which they are initialised and run in the sequence diagram, with respect to the scheduler. The length of the bar on the lifeline for each task is dependent on the frequency at which the function is called. Lower the frequency at which it is called in (a loop), the lower the active time of the task on the lifeline. And every task in the scheduler is run continuously, essentially in a loop.

As seen in Figure 3 above, the tasks are initialised and run in the order: State Handler, LCD, Keypad, Sensor/Switch, one by one in a scheduled loop. The task that is run the least frequently is the state handler, hence has the highest active time. This task is run every 500ms. The State Handler task controls the alarm LED as well as any state transition that are time sensitive. The next task in the order, LCD, is run every 200ms. This task is responsible for displaying error and status messages to the user. The next two tasks (in order: Keypad, Sensor/Switch) have the same frequency of 100ms. These two tasks are run at the highest frequency as performance is critical here. The keypad is used for the user input and hence must be responsive. As for the Sensor/Switch task, the system needs to be able to detect a sensor activation as well toggle the sensor LED (between red and green) as fast as possible.

Each task after it is run (and the method has finished it's execution cycle) will return to the Scheduler so that the next scheduled task can be run.

5. APPENDIX

Source Code 1: main.cpp

```cpp
#include "main.h"

Timer g_timer;  //Timer object
BufferedSerial g_pc(USBTX, USBRX, 115200);  //Serial object

Scheduler g_scheduler; //Scheduler object

DigitalOut g_alarm_led(ALARM_LED);  //Alarm LED

// system is initially in UNSET state
alarm_state_t g_alarm_state = UNSET_STATE;  //global context to keep track of the system state

KeypadControl g_keypad_control;  //Keypad object

BusOut cols_out(KEYPAD_COLS_OUT); // coloums of the keypad
BusIn rows_in(KEYPAD_ROWS_IN); // read the rows of the keypad

TextLCD g_lcd(LCD_PINS);  //LCD display object

BusOut g_switch_cs(SWITCH_CS);  //chip select for switches, controlled using SPI
BusIn g_switch_reading(SWITCH_READING); //These two are for switches

// mosi, miso (unused really), sclk
SPI g_sw(LEDS_SPI); //For the LEDS, controlled using SPI

DigitalOut lat(LEDS_LATCH);

int main() {
    g_timer.start();  // start the timer

    // initialise the tasks and the red/green LEDs
    if(INIT_Tasks() == INIT_FAIL) {
        printf("INIT_Tasks() failed :-(");
        while(1);
    }

    INIT_GRLEDs();

    while(1) {
        // run the scheduler
        g_scheduler.Run(chrono::duration_cast<chrono::milliseconds>(g_timer.elapsed_time()).count());
        wait_us(10000);
    }
}
```

Source Code 2: main.h

```cpp
#ifndef MAIN_H
#define MAIN_H

// include all the necessary header files
#include "pin_map.h"
#include "mbed.h"
#include "scheduler.h"
#include "system.h"
#include "tasks.h"
#include "initialisation.h"
#include "keypad_control.h"
#include "TextLCD.h"

#endif /* MAIN_H */
```

Source Code 3: keypad_control.cpp

```cpp
#include "keypad_control.h"

char KeypadControl::get_key() {
int i,j;
    for (i = 0; i <= 3; i++) {
        cols_out = i;

        // for each bit in rows
        for (j = 0; j <= 3; j++) {

            // if j'th bit of "rows_in" is LOW
            if (~rows_in & (1 << j)) {
                // wait till a key is pressed and released before returning
                while (~rows_in & (1 << j)){}
                return this->keytable[j][3-i];  // return the key pressed from the keytable
            }
        }
    }
    return ' '; // return space if no key is pressed
}
```

Source Code 4: keypad_control.h

```cpp
#ifndef KEYPAD_CONTROL_H
#define KEYPAD_CONTROL_H

#include <cstdint>
#include "mbed.h"

// global variables from main.cpp
extern BusOut cols_out;
extern BusIn rows_in;

class KeypadControl
{
private:
// keytable to map the keys on the keypad
const char keytable[4][4] = {
    {'1', '2', '3', 'F'},
    {'4', '5', '6', 'E'},
    {'7', '8', '9', 'D'},
    {'A', '0', 'B', 'C'}
};

public:
char get_key(); // function to get the key pressed on the keypad

};

#endif /* KEYPAD_CONTROL_H */
```

Source Code 5: initialisation.cpp

```cpp
#include "initialisation.h"

INIT_status INIT_Tasks(void) {
    int8_t rv = 0;  // return value from scheduler functions
    // schedule tasks in the scheduler
    rv |= g_scheduler.schedule_function(state_handler, "led", 1000, ALARM_LED_MS);
    rv |= g_scheduler.schedule_function(lcd_display, "lcd", 1000, LCD_REFRESH_MS);
    rv |= g_scheduler.schedule_function(enter_code, "keypad", 1000, KEYPAD_POLL_MS);
    rv |= g_scheduler.schedule_function(read_switches, "switch", 1000, SWITCH_POLL_MS);

    // check if any of the tasks failed to schedule
    if(rv == -1)
        return INIT_FAIL;
    else
        return INIT_SUCCESS;
}

// Initialise the red/green LEDs from the mbed board
INIT_status INIT_GRLEDs(void) {
    lat = 0;
    g_sw.format(16,0);
    g_sw.frequency(1000000);
    return INIT_SUCCESS;
}
```

Source Code 6: initialisation.h

```cpp
#ifndef INITIALISATION_H
#define INITIALISATION_H

#include "scheduler.h"
#include "tasks.h"
#include "system.h"

// enum to return the status of the initialisation functions
enum INIT_status {
    INIT_FAIL,
    INIT_SUCCESS,
};

// global variables from main.cpp
extern Scheduler g_scheduler;
extern SPI g_sw;
extern DigitalOut lat;

// initialisation functions
INIT_status INIT_Tasks(void);
INIT_status INIT_GRLEDs(void);

#endif /* INITIALISATION_H */
```

Source Code 7: system.h

```cpp
#ifndef SYSTEM_H
#define SYSTEM_H

#include <map>
#include <string>

/* System configuration related stuff goes here */

// refresh rates for the tasks
constexpr int ALARM_LED_MS = 500;
constexpr int LCD_REFRESH_MS = 200;
constexpr int KEYPAD_POLL_MS = 100;
constexpr int SWITCH_POLL_MS = 100;

// alarm system states
typedef enum ALARM_SYSTEM_STATE {
    UNSET_STATE,
    EXIT_STATE,
    SET_STATE,
    ENTRY_STATE,
    ALARM_STATE,
    REPORT_STATE
} alarm_state_t;

//map the alarm state to strings, and is a static variable
static std::map<alarm_state_t, std::string> alarm_state_map = {
    {UNSET_STATE, "UNSET STATE"},
    {EXIT_STATE, "EXIT STATE"},
    {SET_STATE, "SET STATE"},
    {ENTRY_STATE, "ENTRY STATE"},
    {ALARM_STATE, "ALARM STATE"},
    {REPORT_STATE, "REPORT STATE"}
};

static const std::string password = "1234";  // password for the alarm system

constexpr int EXIT_INTERVAL_MS = 60000;  // exit interval in milliseconds

constexpr int ENTRY_INTERVAL_MS = 60000;  // entry interval in milliseconds

constexpr int ALARM_LED_ON_INTERVAL_MS = 120000; // alarm led on interval in milliseconds (for alarm state)

/**< Set to true to show main top level logic debug output on Serial */
#define  SYS_DEBUG_APP_LOGIC    false
#if SYS_DEBUG_APP_LOGIC
    #define debug_printf(...)    printf(__VA_ARGS__)
#else
    #define debug_printf(...)
#endif

#endif /* SYSTEM_H */
```

Source Code 8: tasks.h

```cpp
#ifndef TASKS_H
#define TASKS_H

#include "mbed.h"
#include "system.h"
#include "TextLCD.h"
#include "keypad_control.h"
#include <vector>

// global variables from main.cpp
extern DigitalOut g_alarm_led;
extern alarm_state_t g_alarm_state;
extern TextLCD g_lcd;
extern KeypadControl g_keypad_control;
extern Timer g_timer;

extern BusOut g_switch_cs;
extern BusIn g_switch_reading; //These two are for switches
extern SPI g_sw;
extern DigitalOut lat;

extern BufferedSerial g_pc;

// total number of characters the LCD screen can display in a single line
constexpr int total_no_of_char = 16;

// function prototypes for the scheduled tasks
int state_handler(unsigned long now);
int lcd_display(unsigned long now);
int enter_code(unsigned long now);
int read_switches(unsigned long now);

#endif /* TASKS_H */
```

Source Code 9: task.cpp

```cpp
#include "tasks.h"

std::string input_buffer = "____"; // buffer to store the input from the keypad

// buffer to store the top and bottom line of the lcd display
std::string top_lcd_line_buffer = "\0";
std::string bottom_lcd_line_buffer = "\0";

uint8_t incorrect_attempts_counter = 0; // counter to keep track of the number of incorrect attempts

uint8_t stored_error_value = 0; // store the error value when the alarm is triggered (switches)

// reset the input buffer
void reset_input_buffer() {
    input_buffer = "____";
}

static bool is_alarm_led_on = true; // flag to keep track of the alarm led state
bool reset_previous_time = false; // reset the previous time in the alarm state

// set the initial alarm state conditions
void set_intial_alarm_state() {
    g_alarm_led = 1;
    is_alarm_led_on = true;
    reset_previous_time = true;
}

bool reset_exit_state_previous_time = false; // reset the previous time in the exit state

bool reset_entry_state_previous_time = false; // reset the previous time in the entry state

uint8_t switches = 0; // read the switches value (polling)

bool has_entered_entry_state = false; // flag to keep track when the system has entered the entry state

int state_handler(unsigned long now) {
    switch (g_alarm_state) {
        case UNSET_STATE:
            g_alarm_led = 0;  // turn off the alarm led
            break;
        case ALARM_STATE:
            static long previous_time = now;

            // reset the previous time in the alarm state
            if (reset_previous_time == true) {
                previous_time = now;
                reset_previous_time = false;
            }

            // non blocking delay to turn off the alarm led after a certain interval
            if(now - previous_time >= ALARM_LED_ON_INTERVAL_MS && is_alarm_led_on) {
                g_alarm_led = !g_alarm_led;
                previous_time = now;
                is_alarm_led_on = false;
            }

            break;
        case EXIT_STATE:

            if (switches > 0) {
                set_intial_alarm_state(); // set the initial alarm state conditions
                stored_error_value = switches;
                g_alarm_state = ALARM_STATE; // change the state to alarm state
                break;
            }

            static long exit_previous_time = now;

            if (reset_exit_state_previous_time == true) {
                exit_previous_time = now;
                reset_exit_state_previous_time = false;
            }

            // non blocking delay to change the state to set state after a certain interval
            // and if no switches are activated
            if(now - exit_previous_time >= EXIT_INTERVAL_MS) {
                if(switches == 0)
                    g_alarm_state = SET_STATE;
                exit_previous_time = now;
            }

            g_alarm_led = !g_alarm_led; // blink the alarm led
            break;
        case SET_STATE:
            g_alarm_led = 0;  // turn off the alarm led

            // if the switches are greater than 128 (first switch is activated)
            if (switches >= 128) {
                reset_entry_state_previous_time = true;
                has_entered_entry_state = true;
                g_alarm_state = ENTRY_STATE;

            // else if the switches are between 64 and 127 (last 7 switches)
            } else if (switches > 0 && switches < 128) {
                set_intial_alarm_state();
                stored_error_value = switches;
                g_alarm_state = ALARM_STATE;
            }
            break;

        case ENTRY_STATE:
            // if just entered the entry state and switches are greater than 0
            if (switches > 0 && has_entered_entry_state == false) {
                set_intial_alarm_state();
                stored_error_value = switches;
                g_alarm_state = ALARM_STATE;
                break;
            }

            // if already in entry state and switches are greater than 128
            if (switches > 128 && has_entered_entry_state == true) {
                set_intial_alarm_state();
                stored_error_value = switches;
                g_alarm_state = ALARM_STATE;
                break;
            }
            // door is closed after in entry state
            if(switches == 0 && has_entered_entry_state == true) {
                has_entered_entry_state = false;
            }

            static long entry_previous_time = now;

            if (reset_entry_state_previous_time == true) {
                entry_previous_time = now;
                reset_entry_state_previous_time = false;
            }

            // non blocking delay to change the state to set state after a certain interval
            if(now - entry_previous_time >= ENTRY_INTERVAL_MS) {
                set_intial_alarm_state();
                stored_error_value = 0;
                g_alarm_state = ALARM_STATE;
                break;
            }

            g_alarm_led = !g_alarm_led; // blink the alarm led
            break;

        case REPORT_STATE:
            g_alarm_led = 0; // turn off the alarm led
            break;
        default:
            break;
    }
    return 1;
}

// utility function to get the triggered zones by bitshifting the stored error value
std::string get_triggered_zones() {
    std::vector<int> indices;
    for (unsigned int i = 0; i < 8; i++) {
        if (stored_error_value & (1 << i)) {
            indices.push_back(i+1);
        }
    }
    // convert vector to a comma separated string
    std::string indices_str = "";
    for (unsigned int i = 0; i < indices.size(); i++) {
        indices_str += std::to_string(indices[i]);
        if (i != indices.size() - 1) {
            indices_str += ",";
        }
    }
    // return the string
    if(indices_str == "")
        return "Invalid Code";
    else
        return indices_str;
}

// switch the alarm state based on the password entered
void keypad_state_switch(bool is_password_correct) {
    if(is_password_correct) {
       if(g_alarm_state == UNSET_STATE) {
            reset_exit_state_previous_time = true;
            g_alarm_state = EXIT_STATE;
        } else if(g_alarm_state == ALARM_STATE) {
            g_alarm_state = REPORT_STATE;
        } else if(g_alarm_state == EXIT_STATE || g_alarm_state == ENTRY_STATE) {
            g_alarm_state = UNSET_STATE;
        } else if (g_alarm_state == REPORT_STATE) {
            g_alarm_state = UNSET_STATE;
        }
    } else {
        if(g_alarm_state == UNSET_STATE || g_alarm_state == EXIT_STATE || g_alarm_state == ENTRY_STATE) {
            set_intial_alarm_state();
            stored_error_value = 0;
            g_alarm_state = ALARM_STATE;
        }
    }
}

// utility function to count the number of digits in a string
uint8_t no_of_digits_in_string(std::string str) {
    int count = 0;
    for(unsigned int i = 0; i < str.length(); i++) {
        if(isdigit(str[i]))
            count++;
    }
    return count;
}

// utility function to replace the digits in a string with '*'
```

```
std::string replace_with_asterisk(std::string str) {
    for(unsigned int i = 0; i < str.length(); i++) {
        if(isdigit(str[i]))
            str[i] = '*';
    }
    return str;
}

// update the bottom lcd line buffer based on the alarm state
void bottom_lcd_line_buffer_update() {
    if(g_alarm_state != REPORT_STATE) {
        int count = no_of_digits_in_string(input_buffer);
        if(count == 0) {
            bottom_lcd_line_buffer = "\0";
        } else if (count < 4) {
            bottom_lcd_line_buffer = replace_with_asterisk(input_buffer);
        } else if (count == 4) {
            bottom_lcd_line_buffer = "Press B to set";
        }
    } else {
        bottom_lcd_line_buffer = "Press C to clear";
    }
}

int enter_code(unsigned long now) {
    static int input_buffer_index = 0;
    char key = ' ';
    int code = 0;
    if(g_alarm_state != SET_STATE) {
        key = g_keypad_control.get_key();
        if(key != ' ' && isdigit(key)) {
            //the key gets shifted into the input buffer from right to left
            printf("key: %c\n", key);
            if(input_buffer_index < 4)
                input_buffer[input_buffer_index++] = key;
        } else if(key == 'C') {
            // delete the last character in the input buffer and replace it with '_'
            if (g_alarm_state == REPORT_STATE) {
                keypad_state_switch(true);
            }
            else if (input_buffer_index > 0) {
                input_buffer[--input_buffer_index] = '_'; // replace the last character with '_'
            }
        } else if(key == 'B') {
            //check if there are any '_' in the input buffer
            if(input_buffer.find('_') != std::string::npos) {
                printf("incomplete code\n");
            } else {
                input_buffer_index = 0; // reset the input buffer index
                printf("complete code\n");
                code = std::stoi(input_buffer);
                printf("code: %d\n", code);
                if (code == atoi(password.c_str())) {
                    keypad_state_switch(true);
                    reset_input_buffer(); // reset the input buffer if correct password
                    incorrect_attempts_counter = 0; // reset the incorrect attempts counter
                } else {
                    reset_input_buffer();
                    if(g_alarm_state != ENTRY_STATE && g_alarm_state != ALARM_STATE)
                        incorrect_attempts_counter++; // increment the incorrect attempts counter
                    if(incorrect_attempts_counter == 3) {
                        keypad_state_switch(false);
                        reset_input_buffer();
                        incorrect_attempts_counter = 0; // reset the incorrect attempts counter
                    }
                }
            }
        }
    }
    return 1;
}

// update the top lcd line buffer based on the alarm state and the number of incorrect attempts
// along with calculating the number of spaces to be added to the end of the alarm state string
void top_lcd_line_buffer_update() {
    if(g_alarm_state != REPORT_STATE) {
        unsigned int length = alarm_state_map[g_alarm_state].length();
        // calculate the number of spaces to be added to the end of the alarm state string
        unsigned int no_of_spaces = (total_no_of_char - length) - 2;
        // add the spaces to the end of the alarm state string
        top_lcd_line_buffer = alarm_state_map[g_alarm_state] + std::string(no_of_spaces, ' ');
        if (g_alarm_state == UNSET_STATE || g_alarm_state == EXIT_STATE) {
            top_lcd_line_buffer = alarm_state_map[g_alarm_state] + std::string(no_of_spaces, ' ') +
            std::string("x") + std::to_string(incorrect_attempts_counter);
        }
    } else {
        top_lcd_line_buffer = "E:" + get_triggered_zones();
    }
}

// update the lcd display based on the top and bottom lcd line buffers
int lcd_display(unsigned long now) {
    top_lcd_line_buffer_update();
    bottom_lcd_line_buffer_update();

    g_lcd.cls();
    g_lcd.locate(0, 0);
    g_lcd.printf("%s", top_lcd_line_buffer.c_str());

    g_lcd.locate(0, 1);
    g_lcd.printf("%s", bottom_lcd_line_buffer.c_str());
    return 1;
}

// convert the switch value to the led combination value for the red/green leds
int led_convert(int switch_val) //takes parameter the switch value calculated from the teachers switch code
{
    int led_out = 0; //led output value
    for(int i = 7; i>=0; i--) {
        led_out = (led_out << 2) + 2 - ((switch_val>>i) & 1);
    }
    return led_out; //returns the led_out for led combination value.
}

// task to read the switches value using polling and SPI
int read_switches(unsigned long now) {
    g_switch_cs = 4;
    switches = g_switch_reading;
    g_switch_cs = 5;
    switches= (switches << 4) + g_switch_reading;
    printf("Switches = %d\r\n", switches);

    g_sw.write( ( led_convert(switches) ) );
    lat = 1;
    lat = 0;
    return 1;
}
```

Source Code 10: scheduler.cpp

```
#include "scheduler.h"

Scheduler::Scheduler()
{
    items_in_queue = 0; // initialise variables from the constructor
    queue_start = 0;
    queue_end = 0;
}

int Scheduler::schedule_function(queued_function func, const char * id, unsigned long initial_run, unsigned long recur)
{
  int rv = 0;

    if(strlen(id) > this->MAX_ID_LENGTH) // check if the id is too long
    {
        rv = -1;
    } else {
        // create a new queue item and fill it with the function pointer, next run time, recur time and id
      queue_item new_item;
      new_item.f_ptr = func;
      memset(new_item.item_name, 0, 8);
      memcpy(new_item.item_name, id, strlen(id));
      new_item.recur = recur;
      new_item.next = initial_run;

      rv = add_to_queue(new_item); // add the new item to the queue
    }

  return rv;
}

int Scheduler::schedule_remove_function(const char * id)
```

```
{
 queue_item target;
  int rv = -1;
  // loop through the queue and remove the item with the matching id
  for (unsigned int i = 0; i < items_in_queue; ++i)
  {
    if(queue_get_top(target) == 0)
    {
      if(strcmp(target.item_name, id) == 0)
      {
        rv = 0;
      } else {
        add_to_queue(target);
      }
    } else {
      rv = -1;
      break;
    }
  }

  return rv;
}

int Scheduler::schedule_change_function(const char * id, unsigned long next_run_time, unsigned long new_recur)
{
 queue_item target;
  int rv = -1;
  // loop through the queue and change the next run time and recur time of the item with the matching id
  for (unsigned int i = 0; i < items_in_queue; ++i)
  {
    if(queue_get_top(target) == 0)
    {
      if(strcmp(target.item_name, id) == 0)
      {
        target.next = next_run_time;
        target.recur = new_recur;
        rv = 0;
      }
      add_to_queue(target);
    } else {
      rv = -1;
      break;
    }
  }

  return rv;
}

int Scheduler::Run(unsigned long now)
{
 queue_item target;
  int rv = 0;
  // if there are no items in the queue, return an error code
  if(items_in_queue == 0)
  {
    rv = -1;
  }
  // loop through the queue and run the functions that are due
  for (unsigned int i = 0; i < items_in_queue; ++i)
  {
    if(queue_get_top(target)==0)
    {
      if(target.next <= now)
      {
        int trv;
        trv = (target.f_ptr)(now);
        if(trv == 0)
        {
          rv++;
        }
        if(target.recur != 0)
        {
          target.next = now + target.recur;
          add_to_queue(target);
        }
      } else {
        add_to_queue(target);
      }
    } else {
      rv = -1;
      break;
    }
  }

```

```
    return rv;
}

int Scheduler::queue_get_top(queue_item &item)
{
  int rv = 0;
    //Remove the top item, stuff it into item
    if (queue_end != queue_start) {
            queue_item temp_queue_item = schedule[queue_start];
            queue_start = (queue_start + 1) % queue_schedule_size;
            item = temp_queue_item;
            items_in_queue--;
    } else {
    //if the buffer is empty, return an error code
        rv = -1;
    }

    return rv;
}

int Scheduler::add_to_queue(queue_item item)
{
  //circular buffer is used to store the scheduled functions
  int rv = 0;
    if ((queue_end + 1) % queue_schedule_size != queue_start) {
        schedule[queue_end] = item;
        queue_end = (queue_end + 1) % queue_schedule_size;
        items_in_queue++;
    } else {
        //if buffer is full, error
        rv = -1;
    }
    return rv;
}
```

Source Code 11: scheduler.h

```cpp
#ifndef SCHEDULER_H
#define SCHEDULER_H

#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <mbed.h>

typedef int (*queued_function)(unsigned long); // function pointer type

#define queue_schedule_size 11  // max number of items in the queue

// struct to hold the function pointer and the time to run
struct queue_item {
  queued_function f_ptr;
  unsigned long next;
  unsigned long recur;
  char item_name[8];
};

class Scheduler
{
private:
  unsigned int queue_start;
  unsigned int queue_end;
  unsigned int items_in_queue;
  const uint8_t MAX_ID_LENGTH = 7; // max length of the id string
  queue_item schedule[queue_schedule_size]; // array to hold the scheduled functions

  int queue_get_top(queue_item &item);  // get the top item from the queue
  int add_to_queue(queue_item item);  // add an item to the queue

public:
  Scheduler(); // constructor

  // function to schedule a function to run at a certain time
  int schedule_function(queued_function func, const char * id, unsigned long initial_run, unsigned long recur);
  // function to remove a function from the schedule
  int schedule_remove_function(const char * id);
  // function to change the time and recur time of a scheduled function
  int schedule_change_function(const char * id, unsigned long next_run_time, unsigned long new_recur);
  // start the scheduler
  int Run(unsigned long now);

};

#endif /* SCHEDULER_H */
```

Source Code 12: pin_map.h

```
#ifndef PIN_MAP_H
#define PIN_MAP_H

//Pin definitions go here

/*
                                +----------------------------+
                                |[GND]                 [VOUT]|
                                |[VIN]                   [VU]|
                                |[VB]                   [IF-]|
                                |[NR]                   [IF+]|
                     LEDS_SPI - |[P5]                   [RD-]|
                     LEDS_SPI - |[P6]                   [RD+]|
                     LEDS_SPI - |[P7]                   [TD-]|
                   LEDS_LATCH - |[P8]                   [TD+]|
                                |[P9]                    [D-]|
                                |[P10]                   [D+]|
SWITCH_READING/KEYPAD_ROWS_IN - |[P11]                  [P30]|
SWITCH_READING/KEYPAD_ROWS_IN - |[P12]                  [P29]|
SWITCH_READING/KEYPAD_ROWS_IN - |[P13]                  [P28]|
SWITCH_READING/KEYPAD_ROWS_IN - |[P14]                  [P27]|
                       LCD_RS - |[P15]                  [P26]| - KEYPAD_COLS_OUT/SWITCH_CS
                        LCD_E - |[P16]                  [P25]| - KEYPAD_COLS_OUT/SWITCH_CS
                      LCD_DB4 - |[P17]                  [P24]| - KEYPAD_COLS_OUT/SWITCH_CS
                      LCD_DB5 - |[P18]                  [P23]|
                      LCD_DB6 - |[P19]                  [P22]|
                      LCD_DB7 - |[P20] [L1][L2][L3][L4] [P21]|
                                +----------------------------+
                                     ALARM_LED
*/

#define ALARM_LED       LED1
#define KEYPAD_COLS_OUT p26, p25, p24
#define KEYPAD_ROWS_IN  p14, p13, p12, p11
#define LCD_PINS        p15, p16, p17, p18, p19, p20
#define SWITCH_CS       p26, p25, p24
#define SWITCH_READING  p14, p13, p12, p11
#define LEDS_SPI        p5, p6, p7
#define LEDS_LATCH      p8

#endif /* PIN_MAP_H */
```

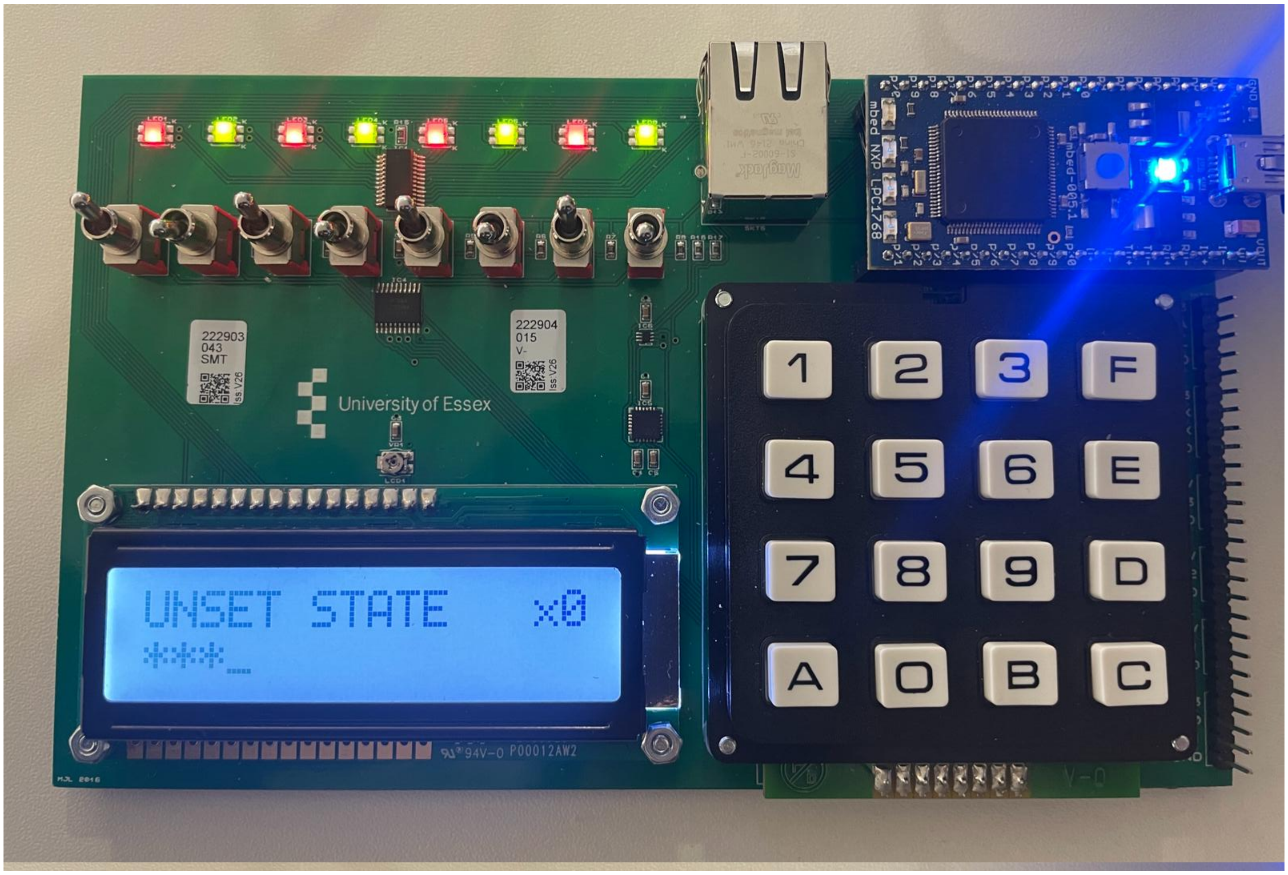

FIG. 4: *Hardware used for the project*